Improving I/O Forwarding Throughput with Data Compression

Benjamin Welton*, Dries Kimpe†, Jason Cope*, Christina M. Patrick‡, Kamil Iskra*, Robert Ross*
*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA
Email: {welton, copej, iskra, rross}@mcs.anl.gov
†Computation Institute, University of Chicago, Chicago, IL 60637, USA
Email: dries@uchicago.edu
‡Pennsylvania State University, University Park, PA 16802, USA
Email: patrick@cse.psu.edu

***Abstract*—While network bandwidth is steadily increasing, it is doing so at a much slower rate than the corresponding increase in CPU performance. This trend has widened the gap between CPU and network speed. In this paper, we investigate improvements to I/O performance by exploiting this gap. We harness idle CPU resources to compress network data, reducing the amount of data transferred over the network and increasing effective network bandwidth. We created a set of data compression services within the I/O Forwarding Scalability Layer. These services transparently compress and decompress data as it is transferred over the network. We studied the effect of the data compression services on a variety of data sets and conducted experiments on a high-performance computing cluster.**

***Keywords*-I/O Forwarding, Scientific Data Compression**

I. Introduction

Computational performance has historically outpaced the development of network interconnects and topologies. As core counts per compute node have increased, the amount of network bandwidth available per compute node has risen but not at a fast enough pace to keep up with the increasing computational performance of the node. This gap is expected to widen as we enter into the era of exascale computing with supercomputers containing many hundreds of computational cores per compute node. In this new era, interconnect bandwidth between nodes and various file systems will be at a premium. By trading CPU performance for a reduction in data size, the effective network bandwidth can be increased.

Interconnect bandwidth typically drops as the distance between nodes increases. For many supercomputers, the total bandwidth between compute nodes is many times larger than the bandwidth to and from external systems. These external systems are often large, shared file systems. For example, for both the IBM Blue Gene/P and the Cray XT series supercomputers, only a limited number of nodes (often called *I/O nodes*) have a direct connection to the external network. This means that all I/O flows through these nodes, creating a bottleneck in reaching the external systems. Since there are often tens to hundreds times more compute nodes than I/O nodes, it is clear that a very small number of compute nodes can easily saturate the network links going to the I/O nodes. As many scientific applications show an I/O pattern biased towards writing (often caused by checkpointing), compressing data before sending it to the I/O nodes reduces the load on the network. Even if the I/O nodes need to decompress the data before sending it on the external network, compression can still be advantageous as many compression algorithms are asymmetric with respect to throughput; decompression is often an order of magnitude faster than compression, ensuring that decompressing data using a limited number of I/O nodes does not become a bottleneck.

IOFSL (I/O Forwarding Scalability Layer), further described in Section III-A, is a portable I/O forwarding implementation. I/O forwarding is used to forward I/O operations from the compute nodes, through the I/O nodes, to the external file system. Due to its presence on both the I/O nodes and the compute nodes, IOFSL is superbly placed to study the compression of I/O data. By adding compression to the I/O forwarding layer, we can easily evaluate its effect on the I/O throughput without modifications to the application or the external file system.

In this paper, we present our recent analysis of how compressing I/O traffic can impact application I/O throughput. In Section II, we present research related to our work. Section III describes the data compression services and the integration with IOFSL. In Section IV, we evaluate our approach using a combination of scientific data sets and synthetic data. Section V describes our conclusions and future work.

II. Related Work

Compression [1] has been explored in many scientific domains. It has been widely used to minimize disk utilization, reduce bandwidth consumption on networks, and reduce energy consumption in hardware. It has found widespread use in the multimedia domain where images and movies are compressed to save disk space and network transmission time [2], [3].

Compression has been used extensively in wireless networks, such as 3G networks, and for website optimizations to reduce the end-to-end transmission time [4], [5], [6], [7], [8], [9], [10]. In these cases, compression reduced transfer

latency and improved response times. Our work differs from the above web-based compression studies. We study the effects of compression in the I/O forwarding layer used in high-performance computing applications. We investigate the effects of data compression on network transmission time as well as its effect on I/O completion time. We show that compression improves network end-to-end transmission time, network bandwidth, and consequently the file I/O throughput.

There have been many studies on the effects of compression to minimize energy consumption [11], [12], [13], [14], [15]. The work presented in these studies focuses on using compression in hardware to reduce the energy consumption of CPU caches and the network. We focus on software compression in high-performance computing clusters to reduce communication latency, improve network bandwidth, and increase the performance of the I/O forwarding layer.

A recent study investigated the effects of compression on power usage in MapReduce clusters [16]. This study focused on increasing I/O performance in order to reduce cluster power consumption. Our work focuses on improving I/O performance to achieve faster time-to-science. Another data compression study was recently performed using BlobSeer [17], a highly parallel distributed data management service. This research differs from our work in that BlobSeer is designed for use in Grid computing environments, while our work is intended for use in high-performance computing systems.

III. HPC Data Compression Services

While data compression methods can decrease the storage footprint of scientific data, generic data compression services for file I/O are not readily available. When these services are available, they are directly integrated into applications [18], [19], data management tools [20], [21], and data storage services [22], [23], [24], [25]. Reintegration of these data compression services into the I/O forwarding layer of the HPC I/O software stack provides the same services in a more portable and transparent manner than through direct integration into HPC software. In this section, we describe I/O forwarding in cluster computing environments, characterize the viable data compression methods for use by HPC applications, and discuss how these data compression methods integrate with I/O forwarding to provide portable and transparent data compression services.

A. I/O Forwarding Overview

The goal of I/O forwarding is to bridge cluster compute nodes with cluster storage systems. On some platforms, storage devices cannot be accessed directly from the compute nodes and I/O forwarding is required to enable such access. I/O forwarding infrastructure can also optimize and reorganize application I/O patterns since it has direct access to all application I/O requests. The I/O Forwarding Scalability Layer (IOFSL) [26], [27], [28] is an example I/O forwarding infrastructure we have used to explore how to bridge the physical storage constraints of HPC systems and optimize application file I/O patterns.

IOFSL consists of two components: a client and a server. When the client receives an I/O request from the application, it does not execute it locally. Instead, it forwards the operation to the server running on a remote, dedicated node. The client integrates with several HPC mechanisms that implement typical HPC file I/O interfaces. A ROMIO driver for IOFSL translates MPI-IO requests into IOFSL client requests. FUSE and sysio IOFSL clients provide a POSIX I/O interface.

The IOFSL server receives and executes the I/O requests forwarded by the clients. After receiving a request, the server decodes it and manages the execution of the I/O operation using a state machine. The state machine executes and communicates with the client to retrieve any additional data required by the I/O request and communicates the results of the I/O operation to the client. The use of a state machine allows for greater concurrency within the server while reducing the number of active threads.

Since IOFSL acts as a transparent intermediary between applications and file systems, it is an ideal place to provide auxiliary data management services, such as data compression services. Identifying and characterizing the behavior of data compression services across a sample of HPC data sets is required to understand the costs and benefits of these services.

B. Data Compression Services

We evaluated three data compression methods for use in our data compression services. The algorithms used in these methods place different emphasis on the compression speed (time to compress data) and the compression ratio (the reduction factor of the compressed data). The Lempel-Ziv-Oberhumer (LZO) compression library is a portable and lossless compression library that focuses on compression speed rather than data compression ratios [29]. The bzip2 compression library also provides portable and lossless compression capabilities and is well known for its compression ratio performance [30]. The popular zlib compression library [31] provides lossless data compression based on the DEFLATE compression algorithm [32]. We evaluated zlib because its compression ratio and speed provide a good balance between the extremes of LZO and bzip2.

We did not investigate floating point compression methods. While such compression methods may improve the performance for some HPC application data sets, we limited our investigation to general purpose data compression libraries so that we could demonstrate the viability of data compression services across a breadth of data types.

C. Integrating Data Compression Services with IOFSL

We developed data compression services for IOFSL based on the LZO, bzip2, and zlib compression libraries for IOFSL. Applications using these services require no modifications as the data compression services are transparent to the end-user. IOFSL provides several scalable mechanisms to optimize potentially expensive HPC file I/O operations. These capabilities can help applications sustain acceptable levels of file I/O performance when using the data compression services.

The data compression services are integrated into the IOFSL client and server network streams. As clients and servers issue I/O requests, the data for these requests are passed to a stream-based network layer. This stream packages the data and transmits them when the stream buffer is full or when it is explicitly flushed. These streams can be stacked so that file I/O data passes through several layers of streams before it is transmitted across the network. The data compression services are implemented within a network stream layer. Once the data is compressed, it is forwarded to a lower-level network stream and is eventually transferred across the network and decompressed by the receiver.

IV. Data Compression Services Evaluation

We evaluated our data compression services within IOFSL. In this section, we describe how IOFSL and the data compression services were deployed in a cluster computing environment, we describe the data sets used in our evaluation, we outline the setup of our data compression experiments, and we analyze the performance and effectiveness of the data compression services.

A. Evaluation Platform and Deployment

We evaluated our data compression and I/O forwarding tools on the Fusion cluster at Argonne National Laboratory. Fusion is a 320 node Linux cluster with a peak performance of 25.9 Tflops. Each Fusion compute node consists of two quad-core, 2.53 GHz Intel Nehalem processors. Fusion has two memory configurations for its compute nodes: 16 fat nodes with 96 GB of RAM and 304 regular nodes with 36 GB of RAM. The compute nodes are connected through two separate networks: a high-performance QDR Infiniband network and a 1-Gbit Ethernet management network. Fusion provides users with several storage services. GPFS and PVFS2 file systems are provided for cluster-wide, high-performance file I/O. Each compute node also provides a local scratch file system using an internal disk drive and a shared memory RAM disk.

Typically, IOFSL executes on a dedicated cluster I/O node that is responsible for handling all file I/O requests generated by compute nodes. The Fusion cluster does not have I/O nodes. Therefore, we allocated additional compute nodes during our evaluation and reserved those nodes as I/O nodes. These I/O nodes hosted the IOFSL server. The benchmarks used in our evaluations integrated the IOFSL client and communicated directly with the IOFSL server.

Name	Description	Format	Source
Zero	Null data	binary	`/dev/zero`
Text	Nucleotide data	text	European Nucleotide Archive [33]
Bin	Air / sea flux data	binary	NCAR Data Archive [34]
Comp	Tropospheric data	GRIB2	NCAR Data Archive [35]
Rand	Random data	binary	`/dev/random`

Table I: Data sets used for evaluation

B. Target Data Sets

The data sets used in our experiments are representative of typical HPC data formats from a variety of scientific domains. We chose a breadth of data sets for our evaluation to help illustrate how each compression method performs for various types of scientific data. The data sets used in our evaluation are described in Table I. They include data expected to compress very well (output from `/dev/zero`), data expected to be incompressible (obtained from `/dev/random`), ASCII text data, binary data, and samples from several freely available scientific data sources.

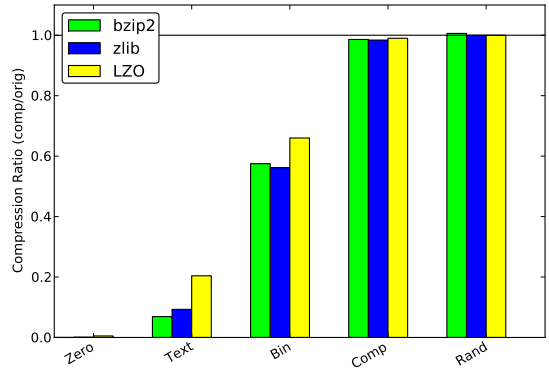
Our initial evaluation explored how well each data set compressed using the LZO, zlib, and bzip2 compression libraries. In this evaluation, we measured the compression ratio, the time to compress, and the time to decompress each data set for each compression method. The results of this evaluation are illustrated in Figure 1.

Figure 1a illustrates the storage or network resources consumed by the compressed data. bzip2 achieves the best compression ratio for most data sets while LZO exhibits the worst. The ratio for the *Text* data set is close to that of the *Zero* set (our best case compression ratio). *Comp* compresses poorly; this data set is in the GRIB2 format and already compressed with JPEG 2000.

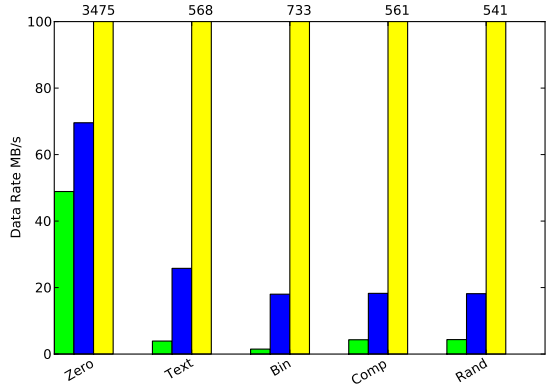
Figures 1b and 1c illustrate the cost of using these algorithms. These figures highlight differences between each compression method and between the compression and decompression operations for each method. Note that the decompression throughput is measured in the volume of *decompressed* data processed per second, same as the compression throughput. For most of the compression methods, the decompression throughput is much greater than the compression throughput (note the different scales on the y axes). LZO achieves the highest throughput rates, confirming its fast, near real-time compression capabilities. bzip2 on the other hand is routinely the slowest compression method. zlib strikes a better balance, achieving compression ratio close to that of bzip2 while maintaining much higher throughput rates.

C. Experimental Design

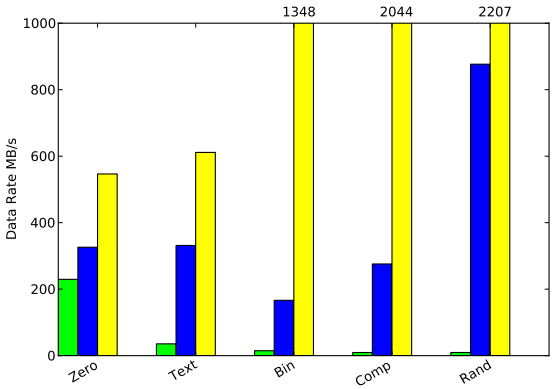
Our experiments evaluated how the IOFSL compression services affected aggregate application I/O throughput. We



(a) Compression Ratio



(b) Compression Throughput



(c) Decompression Throughput

Figure 1: Performance evaluation of the compression algorithms used.

developed a synthetic benchmark to evaluate the compression services for multiple data sets and IOFSL client scaling configurations. The benchmark operates on 128 MB chunks of data from each data set. It executes in two phases. First, it issues compressed read I/O requests. These requests require the IOFSL server to compress the data and the IOFSL client to decompress it. During the second phase, the benchmark issues write I/O requests for the same chunk of data. This requires the IOFSL client to compress the data and the server to decompress it. For each phase, the benchmark reports the aggregate throughput of the data for all processes based on the size of the uncompressed data. This metric illustrates the potential speedup from using the data compression services.

On the Fusion cluster, we used the shared memory RAM disk (/dev/shm) to store the benchmark data. Using the RAM disk in our evaluation limited file system noise and removed the storage system as a bottleneck. In these experiments, a single IOFSL server was deployed on a dedicated node. We executed our synthetic benchmark (using the IOFSL client) at scales ranging from 8 to 256 client processes with 8 clients per compute node.

D. Analysis of Results

Figure 2 shows the observed transfer rate when reading or writing each of the data sets described in Table I. The solid and dashed lines indicate the respective read and write speed when compression is disabled. Any result showing a read speed higher than the solid line indicates that compression increased I/O throughput. While we evaluated each compression method with a wide range of client counts (up to 256) on both Ethernet and Infiniband networks, due to space constraints Figure 2 only shows the results for 128 clients using Ethernet.

It is clear that the results are highly dependent on the compression algorithm. The first graph (top of Figure 2) shows the results for the bzip2 algorithm. For the *Zero* data set, performance is greatly improved for both reads and writes. However, looking at the *Text* data set, read performance drops to a third of the uncompressed case, while write performance doubles. The reason for the different behavior between these two data sets can be found in Figure 1. While *Zero* data set can be decompressed with bzip2 at over 200 MB/s, the text data decompression speed is only a fraction of this. Compressing data using the bzip2 algorithm is even slower: around 50 MB/s for *Zero*, and less than 5 MB/s for *Text*. The fact that bzip2 nevertheless manages to improve write performance is explained by looking at the amount of CPU time available. Since, for writing, compression takes place on the client, an increasing number of clients means more processing time is available for compression. However, on the server, where decompression happens, the amount of processing time remains fixed, regardless of the number of clients being serviced. For both the *Zero* and *Text* data sets, the forwarding server can decompress data fast enough to show an improvement in write speed when comparing to the uncompressed case. However, for reading, when the server is responsible for compressing the data, only *Zero* data set can be compressed fast enough. When compressing text data, at a rate about ten times slower than that of *Zero*, the available CPU time becomes a bottleneck, explaining the drop in read performance. For the other data sets, bzip2's limited throughput is far too low to be usable.

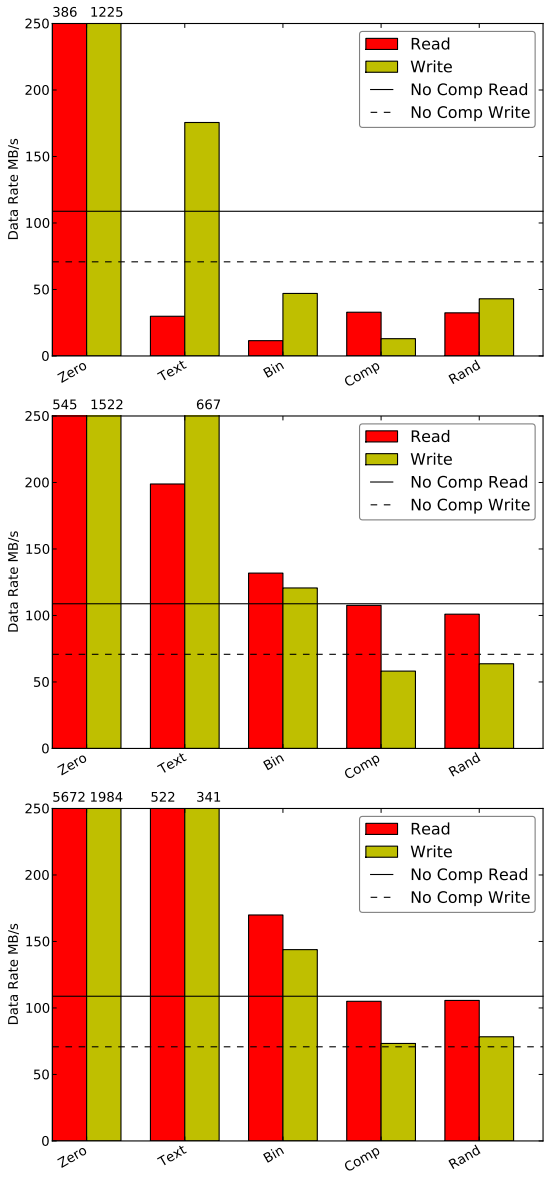


Figure 2: Performance on the Ethernet network with 128 clients, using: bzip2 *(top)*, zlib *(center)*, and LZO *(bottom)*.

LZO (bottom graph of Figure 2), having a very high compression and decompression throughput, delivers the best performance, showing transfer speeds close to the uncompressed case even for hard to compress data. The small drop in read performance can be attributed to the relatively minor overhead caused by the fast LZO compression algorithm. Highly compressible data, such as the *Zero* or *Text* data sets, show a speedup of a factor of 5 to 50 for both reading and writing. For the *Bin* data set, throughput almost doubles.

The zlib results (center graph of Figure 2) are quite similar to those of LZO. Data sets that can be compressed show an appreciable speedup, while hard to compress data shows a larger—albeit still small—reduction in I/O throughput. The increased overhead is directly related to the slower compression and decompression throughput rates of zlib.

Figure 3 shows the I/O throughput our approach would deliver if decompression overhead were negligible. This would be the case if the decompression could be offloaded to a dedicated processor. Assuming an overhead-free decompression is not unrealistic, given the fact that dedicated accelerators capable of compressing and decompressing data using zlib at a rate of multiple hundred megabytes per second are already commercially available.

Avoiding decompression overhead can also be realized by simply not decompressing the data, instead storing the compressed representation on the file system. Overhead-free compression can be obtained by using a sufficiently large number of clients, as—for writes—the total compression throughput scales linearly with the number of clients (and thus processors). For checkpointing, storing the compressed representation would be a reasonable thing to do as checkpoints are always written but rarely read.

The data shown in Figure 3 is obtained by taking the uncompressed I/O throughput (writes), divided by the compression ratio. Therefore, the throughput of hard to compress data, such as the *Comp* data set which has a compression ratio close to 1, is hardly affected.

The *Text* data set however, which has a very small compression ratio, shows dramatic increases in write throughput, ranging from about 1.5 gigabytes per second for LZO to almost 5 gigabytes per second for bzip2. The throughput for the *Bin* data set, which compresses less well, still almost doubles to 600 megabytes per second.

Since for all data sets, throughput is at least that of the uncompressed case, compression can always be safely enabled in this scenario.

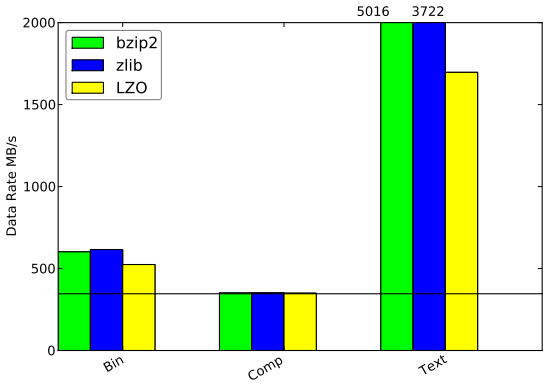


Figure 3: Projected write performance over Infiniband with no compression overhead.

Figure 4 presents a scalability study of the data compression service for different client counts ranging from 8 to 256,

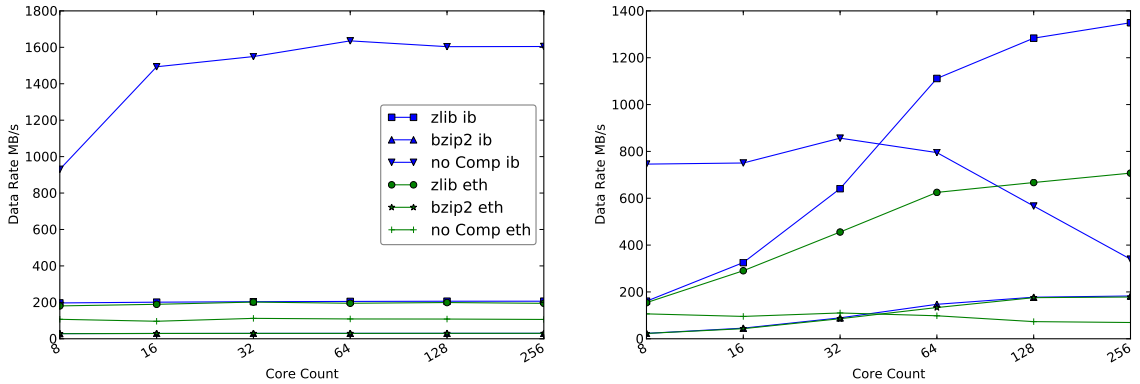


Figure 4: Scalability of data compression service for *Text* with different numbers of clients, using zlib, bzip2, and no compression, over Ethernet and Infiniband networks, when reading *(left)* and writing *(right)*.

with two compression methods (zlib and bzip2), and both Ethernet and Infiniband. The *Text* data set is being used. For any compression method to be useful, its throughput should exceed that of the plain transfer without compression (denoted in the figure as *no Comp*). This is especially hard when reading (left graph), since, as discussed earlier, the I/O forwarding server becomes a bottleneck due to compression overhead. zlib manages to beat the odds when running over the Ethernet network, but not over Infiniband, which has an order of magnitude greater bandwidth. bzip2, due to higher compression overhead, succeeds with neither. The situation is different when writing (right graph), as increasing client parallelism can help hide the cost of compression overhead; we are also helped by the fact that the raw writing performance drops above 64 clients, especially with Infiniband (we are investigating the cause of this). Nevertheless, when writing, zlib compression is advantageous for all client counts tested when using Ethernet, and for counts exceeding 32 when using Infiniband. Even the high cost of bzip2 can be offset, at least on Ethernet, with client counts of 64 and higher.

V. Conclusions and Future Work

In this paper, we studied the feasibility of data compression and decompression while performing I/O. We analyzed common compression algorithms and studied their effect on the overall I/O performance. To evaluate compression algorithm performance on I/O, we selected scientific data obtained from multiple fields for testing. For certain types of scientific data, we observed significant bandwidth improvements. Our work shows that the benefits of compressing data prior to transmission are highly dependent on the data being transferred.

We are pursuing several areas of future work. Instead of decompressing data before storing it on disk, we are investigating storing compressed data, thus avoiding decompression overhead until read time. As many scientific applications exhibit write-heavy I/O patterns, this optimization could have a significant effect on the total execution time.

We are also investigating an adaptive compression approach, in which the best compression algorithm is selected automatically. In cases where compression does not offer any benefits, data compression is temporarily disabled.

In Section IV-D, we studied the performance assuming a zero-overhead compression and decompression algorithm. While not exactly cost-free, dedicated compression accelerators, or coprocessors such as GPUs, have the potential to reduce the compression overhead to that of a simple memory copy. If the data compression and decompression time can be reduced, the overhead associated with trying to compress incompressible data can be avoided, improving the applicability of our approach. In future work, we will investigate compression offloading mechanisms to reduce the cost of the compression methods and validate the projections presented in Section IV-D

The compression algorithms studied in this paper are general purpose. As our work is situated in the I/O layer, these conditions cannot be guaranteed without modifying other layers of the software stack. While there is a wide range of special purpose compression methods, we did not include them in our study due to the stricter requirements these algorithms place on the input data. In future, we plan to relax our requirements and analyze floating point compression algorithms as part of our data compression services.

Acknowledgements

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. The IOFSL project is supported by the DOE Office of Science and National Nuclear Security Administration (NNSA). We gratefully acknowledge the computing resources provided on Fusion, a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Comput. Surv.*, vol. 19, no. 3, pp. 261–296, 1987.

[2] D. Le Gall, "Mpeg: a video compression standard for multimedia applications," *Commun. ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[3] H. Man, A. Docef, and F. Kossentini, "Performance analysis of the jpeg 2000 image coding standard," *Multimedia Tools Appl.*, vol. 26, no. 1, pp. 27–57, 2005.

[4] X. Chen, W. Wang, and G. Wei, "Impact of http compression on web response time in asymmetrical wireless network," in *NSWCTC '09: Proceedings of the 2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, 2009, pp. 679–682.

[5] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, 1997, pp. 181–194.

[6] W. Dong, X. Chen, S. Xu, W. Wang, and G. Wei, "Proxy-based object packaging and compression: a web acceleration scheme for umts," in *WiCOM'09: Proceedings of the 5th International Conference on Wireless communications, networking and mobile computing*, 2009, pp. 4965–4969.

[7] K. J. Ma and R. Bartos, "Analysis of transport optimization techniques," in *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, 2006, pp. 611–620.

[8] A. B. King, *Website optimization*. O'Reilly, 2008.

[9] R. L. R. Mattson and S. Ghosh, "Http-mplex: An enhanced hypertext transfer protocol and its performance evaluation," *J. Netw. Comput. Appl.*, vol. 32, no. 4, pp. 925–939, 2009.

[10] S. Deshpande and W. Zeng, "Scalable streaming of jpeg2000 images using hypertext transfer protocol," in *MULTIMEDIA '01: Proceedings of the ninth ACM international conference on Multimedia*, 2001, pp. 372–381.

[11] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok, "Energy and performance evaluation of lossless file data compression on server systems," in *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009, pp. 1–12.

[12] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, 2004, p. 212.

[13] R. Canal, A. González, and J. E. Smith, "Very low power pipelines using significance compression," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 181–190.

[14] K. C. Barr and K. Asanović, "Energy-aware lossless data compression," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 250–291, 2006.

[15] R. Das, A. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. Yousif, and C. Das, "Performance and power optimization through data compression in network-on-chip architectures," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008, pp. 215–225.

[16] Y. Chen, A. Ganapathi, and R. Katz, "To compress or not to compress-compute vs. IO tradeoffs for MapReduce energy efficiency," in *Proceedings of the first ACM SIGCOMM workshop on Green networking*. ACM, 2010, pp. 23–28.

[17] B. Nicolae, "High throughput data-compression for cloud storage," *Data Management in Grid and Peer-to-Peer Systems*, pp. 1–12, 2011.

[18] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J Comp Phys*, no. 117, pp. 1–19, 1995.

[19] LAMMPS documentation: dump command. [Online]. Available: http://lammps.sandia.gov/doc/dump.html

[20] "Hierarchical data format 5," http://www.hdfgroup.org/HDF5/.

[21] A. Collette. (2009) Lzf compression filter for hdf5. [Online]. Available: http://h5py.alfven.org/lzf/

[22] btrfs wiki. [Online]. Available: https://btrfs.wiki.kernel.org/

[23] M. Larabel. (2010, August) Using disk compression with btrfs to enhance performance.

[24] The hadoop disrtibuted file system. [Online]. Available: http://hadoop.apache.org/hdfs/

[25] Using lzo compression. [Online]. Available: http://wiki.apache.org/hadoop/UsingLzoCompression

[26] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization Techniques at the I/O Forwarding Layer," in *IEEE International Conference on Cluster Computing 2010*, 2010.

[27] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O Forwarding Framework for High–Performance Computing Systems," in *IEEE International Conference on Cluster Computing 2009*, 2009.

[28] J. Cope, K. Iskra, D. Kimpe, and R. Ross, "Bridging HPC and Grid file I/O with IOFSL," in *Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.

[29] LZO real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[30] bzip2. [Online]. Available: http://www.bzip2.org/

[31] zlib. [Online]. Available: http://www.zlib.net/

[32] P. Deutsch. (1996, May) DEFLATE compressed data format specification version 1.3. [Online]. Available: http://tools.ietf.org/html/rfc1951

[33] European Nucleotide Archive. [Online]. Available: http://www.ebi.ac.uk/

[34] CORE.2 global air-sea flux dataset. [Online]. Available: http://dss.ucar.edu/datasets/ds260.2/

[35] NCEP FNL operational model global tropospheric analyses. [Online]. Available: http://dss.ucar.edu/datasets/ds083.2/